WEB DAY 2026

Spec Driven Development

Matteo Ronchi
Software Architect
WorkWave

improove

WEB DAY 2026

Kudos

improove

Hello!

Matteo Ronchi

Software Architect @ WorkWave

25 years in Web Development

github.com/cef62

linkedin.com/in/matteoronchi

AI coding tools are everywhere

MCP Servers · Autonomous agents · AI IDEs · Plugins

More capability = **more need for structure**.

I spent the last year testing SDD frameworks,
building training programs, and shipping production code
with AI agents.

Some things worked. Some didn't.

What is a Spec?

A structured, behavior-oriented document that describes software functionality in natural language.

Not Just a Prompt

A prompt is a **one-shot instruction**
A spec is a **persistent agreement**

```
Prompt: "Add user authentication"

Spec:   Requirements, behaviors, edge cases,
        constraints, testing strategy —
        all in one reviewable artifact.
```

Spec Example

```
## Feature: User Authentication

### Requirements
- Sign up with email/password
- Password: 8+ chars, 1 number, 1 special
- Max 5 failed attempts → 15-min lockout

### Behavior
- Success → redirect to dashboard
- Failure → show error, increment counter
- Lockout → show countdown timer

### Edge Cases
- Expired session during submit
- Concurrent login attempts
```

Spec ≠ Memory Bank

Two different concepts. Both matter.

Memory Bank	Spec
Cross-session context	Task-specific contract
Coding rules, patterns	Feature requirements
Shared across all tasks	Guides one feature
`CLAUDE.md`, `.cursorrules`	`spec.md`, `plan.md`

Memory Banks **feed** your specs.
Specs **focus** your agent.

The SDD Premise

"Specs describe intent. Agents generate code."

You write: WHAT and WHY

The agent writes: HOW

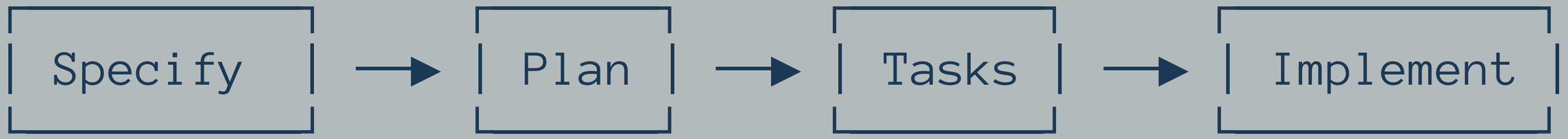
SDD Levels

Birgitta Böckeler (Thoughtworks) defined three adoption levels:

Spec-first → write spec, generate code, maybe discard spec
Spec-anchored → spec evolves with the feature over time
Spec-as-source → humans only edit specs, never touch code

Today, **spec-first** is the only level with mature tooling.

The SDD Workflow



Each stage produces an artifact that guides the next.

Specify: Requirements, behaviors, edge cases
Plan: Architecture decisions, component design
Tasks: Ordered, atomic steps with dependencies
Implement: One task at a time. Review. Commit. Repeat.

Plan Mode ≠ SDD

Every AI coding tool has a plan mode now.
It's useful, but it's not the same thing.

	Plan Mode	**SDD**
Artifact	Ephemeral	Persistent
Scope	Single session	Multi-session
Format	Freeform	Structured
Reviewable	By you, right now	By the team, later

The Tool Landscape

Four tools worth knowing in 2026:

Amazon Kiro — opinionated IDE with built-in SDD
GitHub Spec Kit — open-source, agent-agnostic (74k+)
spec-workflow-mcp — MCP server with web dashboard
Superpowers — methodology framework (84k+)

And one more option: **no framework at all**.

Amazon Kiro

A VS Code fork with the most opinionated SDD workflow.

Requirements → Design → Tasks
Uses EARS notation. Human review gates at every step.

- Great onboarding for SDD concepts
- Can over-engineer small tasks
- Locked to the Kiro IDE

GitHub Spec Kit

Open-source Python MCP. Works with **22+ AI agents**.

Constitution → Specify → Plan → Tasks → Implement

The "Constitution" concept is powerful:
project-level immutable principles that constrain every spec.

- Agent-agnostic
- Rich ecosystem
- Generated specs can be verbose

spec-workflow-mcp

An MCP server – works inside any MCP-compatible tool.

Steering → Specifications → Implementation → Verification

- Real-time web dashboard
- VSCode extension for review and approval
- MCP compatibility required

Superpowers

Not strictly an SDD tool. A **methodology encoded as skills**.

- Plan with atomic tasks
- Git worktree isolation per task
- Subagent-driven implementation with self-review
- TDD enforcement (red-green-refactor)
- Works across Claude Code, Codex, Gemini CLI

What Superpowers Does Right

The framework enforces three ideas:

1. **Plans are small.** Each task = 2-5 minutes of agent work.
2. **Review is mandatory.** Inline self-review catches 3-5 bugs in ~30 seconds.
3. **Isolation is cheap.** Git worktrees mean one branch per task.

I adopted all three as personal practice.
I don't always run Superpowers to do it.

The No-Framework Path

Every modern AI tool already has SDD infrastructure.

Claude Code: `CLAUDE.md` + `/plan` + skills + subagents
Cursor: `.cursorrules` + plan mode + background agents
Copilot: `copilot-instructions.md` + coding agent
Windsurf: `.windsurfrules` + Cascade

You're probably already doing lightweight SDD.
You just haven't formalized it.

CLAUDE.md as SDD Foundation

```
project/
├── CLAUDE.md                ← Project memory bank
├── .claude/
│   └── rules/
│       ├── frontend.md      ← Scoped rules per area
│       └── testing.md
├── specs/
│   └── auth-feature/
│       ├── spec.md          ← Feature spec
│       ├── plan.md          ← Implementation plan
│       └── tasks.md         ← Ordered task list
└── src/
```

No framework. No dependencies. Just markdown and discipline.

Writing Specs That Work

Addy Osmani's principles for AI agent specs:

1. **Be behavior-oriented** — describe what, not how to
2. **Define boundaries** — Always / Ask first / Never
3. **Include edge cases** — the AI might skip them
4. **Keep it atomic** — one concern per section
5. **Write for the agent** — think "Agent Experience," not documentation

More detailed ≠ always better.

Too many directives and the AI wanders.

The Honest Assessment

Not everyone is convinced.

The criticism is worth hearing.

Kent Beck

Emphasizing writing the whole specification before implementation encodes the assumption you won't learn anything during implementation.

Birgitta Böckeler (Thoughtworks)

I'd rather review code than all these markdown files.

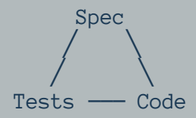
I frequently saw the agent not follow all the spec instructions.

The waterfall question

Is SDD just waterfall repackaged for the AI era?

The SDD Triangle (Drew Breunig)

SDD is not a one-way pipeline.



Spec defines tests and constrains code.
Tests validate code against spec intent.
Code reveals decisions that improve the spec.

It's a feedback loop. Not a waterfall.
Start light. Iterate. Let the spec grow with the code.

When Is SDD Worth It?

Use SDD when

- Multiple contributors need to understand the work
- Requirements must persist beyond your current session
- The task is complex enough to split into sub-tasks
- You're working with an external team or vendor

Skip SDD when

- Solo exploration or learning
- Small fixes (a few hours of work)
- Requirements are still too vague to write down
- You already know the task will pivot mid-development

How I Actually Work

- CLAUDE.md as my memory bank.
- Atomic tasks. Review every diff. Commit per task.

80% of the time
Plan mode → informal spec

Write the plan as a markdown file.
Execute step by step.
Transform into documentation when the feature ships.

20% of the time
Full SDD workflow

For large features, team handoffs, vendor collaboration.
Use **Superpowers** or the no-framework path.

Key Takeaways

1. **SDD adds structure** — but you choose how much
2. **You don't need a framework** — CLAUDE.md + plan mode + discipline gets you 80% there
3. **Specs are communication tools** — not magic, not guarantees
4. **Start with spec-first** — discard specs after shipping, evolve the practice over time

Resources

Articles:

- Böckeler: SDD Tools (martinfowler.com)
- Osmani: How to Write Good Specs
- Breunig: The SDD Triangle

Resources

Tools:

- Amazon Kiro
- GitHub Spec Kit
- spec-workflow-mcp
- Superpowers
- claude-code-spec-workflow

Thank you!

Matteo Ronchi

github.com/cef62
linkedin.com/in/matteoronchi
github.com/cef62/conference-webconf-2026

WEB DAY 2026

Thanks !

improove